METHOD AND SYSTEM USING HARDWARE ASSISTANCE FOR TRACING INSTRUCTION DISPOSITION INFORMATION

CROSS-REFERENCE TO RELATED APPLICATION

The present application is related to the following application: Application Serial Number (Attorney Docket Number AUS920010714US1), filed (TBD), titled "Method and system using hardware assistance for instruction tracing by revealing executed opcode or instruction".

BACKGROUND OF THE INVENTION

1.   Field of the Invention

The present invention relates generally to an improved data processing system and, in particular, to a method and system for instruction processing within a processor in a data processing system.

2.   Description of Related Art

In analyzing the performance of a data processing system and/or the applications executing within the data processing system, it is helpful to understand the execution flows and the use of system resources. Performance tools are used to monitor and examine a data processing system to determine resource consumption as various software applications are executing within the data processing system. For example, a performance tool may identify the most frequently executed modules and instructions in a data processing system, or it may identify those modules which allocate the largest amount

of memory or perform the most I/O requests. Hardware performance tools may be built into the system or added at a later point in time. Software performance tools also are useful in data processing systems, such as

5   personal computer systems, which typically do not contain many, if any, built-in hardware performance tools.

One known software performance tool is a trace tool. A trace tool may use more than one technique to provide trace information that indicates execution flows for an

10   executing program. For example, a trace tool may log every entry into, and every exit from, a module, subroutine, method, function, or system component. Alternately, a trace tool may log the amounts of memory allocated for each memory allocation request and the

15   identity of the requesting thread. Typically, a time-stamped record is produced for each such event. Corresponding pairs of records similar to entry-exit records also are used to trace execution of arbitrary code segments, starting and completing I/O or data

20   transmission, and for many other events of interest.

In order to improve software performance, it is often necessary to determine where time is being spent by the processor in executing code, such efforts being commonly known in the computer processing arts as

25   locating "hot spots." Within these hot spots, there may be lines of code that are frequently executed. When there is a point in the code where one of two or more branches may be taken, it is useful to know which branch is the mainline path, or the branch most frequently

30   taken, and which branch or branches are the exception branches. Grouping the instructions in the mainline

branches of the module closely together also increases the likelihood of cache hits because the mainline code is the code that will most likely be loaded into the instruction cache.

Ideally, one would like to isolate such hot spots at the instruction level and/or source line level in order to focus attention on areas which might benefit most from improvements to the code. For example, isolating such hot spots to the instruction level permits a compiler developer to find significant areas of suboptimal code generation. Another potential use of instruction level detail is to provide guidance to CPU developers in order to find characteristic instruction sequences that should be optimized on a given type of processor.

Another analytical methodology is instruction tracing by which an attempt is made to log every executed instruction. Instruction tracing is an important analytical tool for discovering the lowest level of behavior of a portion of software.

However, implementing an instruction tracing methodology is a difficult task to perform reliably because the tracing program itself causes some interrupts to occur. If the tracing program is monitoring interrupts and generating trace output records for those interrupts, then the tracing program may log interrupts that it has caused through its own operations. In that case, it would be more difficult for a system analyst to interpret the trace output during a post-processing phase because the information for the interrupts caused by the tracing program should first be recognized and then should be filtered or ignored when recognized.

More specifically, instruction tracing may cause interrupts while trying to record trace information because the act of accessing an instruction may cause interrupts, thereby causing unwanted effects at the time

5   of the interrupt and generating unwanted trace output information. A prior art instruction tracing technique records information about the next instruction that is about to be executed. In order to merely log the instruction before it is executed, several interrupts can

10 be generated with older processor architectures, such as the X86 family, while simply trying to access the instruction before it is executed. For example, an instruction cache miss may be generated because the instruction has not yet been fetched into the instruction

15 cache, and if the instruction straddles a cache line boundary, another instruction cache miss would be generated. Similarly, there could be one or two data cache misses for the instruction's operands, each of which could also trigger a page fault.

20 In order to accurately reflect the system flow, the tracing software should not trace its own instructions or the effects of its execution. However, if the tracing software generates interrupts, exceptions, etc., it may be difficult to determine whether the interrupts would

25 occur normally by the software without tracing or if the interrupt is only caused by the act of tracing. For example, if the tracing code is also tracing data accesses, which have not yet occurred, any page faults associated with the access of the data would be generated

30 not only by the act of tracing but also would have occurred when the instruction itself was executed. In

this case, if the tracing software suppresses tracing of the exception, the information regarding the exception would be lost. If the tracing software is attempting to copy an instruction that has not yet been executed,

5   interrupts associated with the act of copying should not be recorded. If the tracing software reads the actual instruction and the instruction passes a page boundary, then normal execution path would cause a page fault, which should be recorded. If the tracing software reads

10   more bytes than is required to execute the instruction and the read operation passes a page boundary, then the normal execution path may or may not pass a page boundary.

In addition to the above-mentioned difficulties,

15   some advanced processor architectures incorporate other features that cause additional difficulties, such as predication, which is the conditional execution of an instruction based on a qualifying predicate. If a processor implements predication functionality, then

20   typically most processor instructions can be guarded by a qualifying predicate, which is typically a predicate register whose value determines whether the processor commits the results computed by the qualified instruction. Predicate registers are usually one-bit

25   values in which a "zero"-valued predicate is interpreted as false and in which a "one"-valued predicate is interpreted as true. If the predicate is true, then the instruction executes completely or fully; if the predicate is false, then the instruction does not execute

30   fully because it does not modify the state of the processor in a way that would affect the execution of

subsequent instructions. In other words, if the predicate is false, then the predicated instruction's architectural updates are suppressed, and the instruction behaves like a so-called "nop", which is an abbreviated

5   term for a "no-op" or a "no operation".

Predication is particularly useful because predicated instructions can be used for conditional execution of branches, which results in longer series of unbranched instruction streams and the elimination of

10   associated mispredict penalties. In essence, predication allows a compiler to convert control dependencies into data dependencies, thereby allowing the compiler to optimize instruction scheduling during compilation. However, predication creates additional difficulties for

15   instruction tracing because the instruction tracing software should be able to identify which of the traced instructions in the trace output data were fully executed.

Therefore, it would be advantageous to have hardware

20   structures within the processor that provide information to assist in the identification of partially executed instructions versus fully executed instructions, i.e., non-committed instructions versus committed instructions, in conjunction with other functionality within the

25   instruction tracing software.

SUMMARY OF THE INVENTION

A method, system, apparatus, or computer program product uses a processor mechanism that generates a disposition indicator that reflects whether an instruction has been partially or fully executed by the processor, i.e., whether the results of the instruction are committed. The disposition indicator is then captured in conjunction with other instruction trace information for subsequent post-processing. For most instructions, a predicate register value controls whether an instruction is partially or fully executed; for these instructions, the disposition indicator equals the value of the predicate register. For other instructions that are not predicated or cannot be predicated, the disposition indicator is set when the instruction is executed. A series of indicators for a series of instructions may be stored in a disposition trace buffer upon the completion of each instruction; the disposition trace buffer may be located within the processor or within memory.

BRIEF DESCRIPTION OF THE DRAWINGS

5   The novel features believed characteristic of the
invention are set forth in the appended claims. The
invention itself, further objectives, and advantages
thereof, will be best understood by reference to the
following detailed description when read in conjunction
10   with the accompanying drawings, wherein:

Figure 1A depicts a typical data processing system in
which the present invention may be implemented;

Figure 1B depicts typical structures in a processor
and a memory subsystem in which the present invention may
15   be implemented;

Figure 1C depicts typical software components within
a computer system illustrating a logical relationship
between the components as functional layers of software;

Figure 1D depicts a typical relationship between
20   software components in a data processing system that is
being analyzed in some manner by a trace facility;

Figure 1E depicts typical phases that may be used to
characterize the operation of a tracing facility;

Figure 2A depicts an executed-instruction register
25   within a processor that may be used to reveal an executed
instruction;

Figure 2B depicts an executed-instruction register
within a processor that is protected by an
executed-instruction (EI) control flag;

Figure 2C depicts a flowchart for the use of a EI control flag associated with an executed-instruction register within a processor;

Figure 2D depicts an executed-instruction register that is protected by an EI flag to be used in conjunction with other control flags, such as interrupt control flags;

Figure 3A depicts a taken-branch instruction buffer to be used to store executed instructions with respect to the most recent branch-type instruction;

Figure 3B depicts a flowchart for the use of a taken-branch instruction buffer within a processor;

Figure 4 depicts an alternative embodiment for a taken-branch instruction buffer to be used to store executed instructions;

Figure 5A is a prior art diagram depicting the predicate register file in the IA-64 processor architecture;

Figure 5B depicts an executed-instruction register and a disposition trace register within a processor that may be used to reveal an executed instruction and its disposition;

Figure 5C depicts a disposition trace register within a processor that is protected by a disposition trace (DT) control flag;

Figure 5D is a flowchart that depicts the use of a DT control flag for a disposition trace register within a processor;

Figure 6A depicts a multi-bit disposition trace register to be used to store disposition information with respect to a series of recently executed instructions;

Figure 6B is a flowchart that depicts the use of a
5   multi-bit disposition trace register within a processor; and

Figure 6C depicts an alternative embodiment for a disposition trace buffer to be used to store disposition information.

10

DETAILED DESCRIPTION OF THE INVENTION

5   The present invention is directed to hardware
structures within a processor that assist tracing
operations. As background, a typical organization of
hardware and software components within a data processing
system is described prior to describing the present
10   invention in more detail.

With reference now to the figures, **Figure 1A** depicts
a typical data processing system in which the present
invention may be implemented. Data processing system **100**
contains network **101**, which is the medium used to provide
15   communications links between various devices and computers
connected together within distributed data processing
system **100**. Network **101** may include permanent
connections, such as wire or fiber optic cables, or
temporary connections made through telephone or wireless
20   communications. In the depicted example, server **102** and
server **103** are connected to network **101** along with storage
unit **104**. In addition, clients **105-107** also are connected
to network **101**. Clients **105-107** may be a variety of
computing devices, such as personal computers, personal
25   digital assistants (PDAs), etc. Distributed data
processing system **100** may include additional servers,
clients, and other devices not shown. In the depicted
example, distributed data processing system **100** may
include the Internet with network **101** representing a
30   worldwide collection of networks and gateways that use the
TCP/IP suite of protocols to communicate with one another.

Of course, distributed data processing system **100** may also be configured to include a number of different types of networks, such as, for example, an intranet, a local area network (LAN), or a wide area network (WAN).

5   **Figure 1A** is intended as an example of a heterogeneous computing environment and not as an architectural limitation for the present invention. The present invention could be implemented on a variety of hardware platforms, such as server **102** or client **107**

10   shown in **Figure 1A**. Requests for the collection of performance information may be initiated on a first device within the network, while a second device within the network receives the request, collects the performance information for applications executing on the second

15   device, and returns the collected data to the first device.

       With reference now to **Figure 1B**, a block diagram depicts typical structures in a processor and a memory subsystem that may be used within a client or server, such

20   as those shown in **Figure 1A**, in which the present invention may be implemented. Hierarchical memory **110** comprises Level 2 cache **112**, random access memory (RAM) **114**, and non-volatile memory **116**. Level 2 cache **112** provides a fast access cache to data and instructions

25   that may be stored in RAM **114** in a manner which is well-known in the art. RAM **114** provides main memory storage for data and instructions that may also provide a cache for data and instructions stored in nonvolatile memory **116**, such as a flash memory or a disk drive.

Processor **120** comprises a pipelined processor
capable of executing multiple instructions in a single
cycle. During operation of the data processing system,
instructions and data are stored in hierarchical memory

5 **110**. Data and instructions may be transferred to
processor **120** from hierarchical memory **110** on a common
data path/bus or on independent data paths/buses. In
either case, processor **120** may provide separate
instruction and data transfer paths within processor **120**

10 in conjunction with instruction cache **122** and data cache
124. Instruction cache **122** contains instructions that
have been cached for execution within the processor.
Some instructions may transfer data to or from
hierarchical memory **110** via data cache **124**. Other

15 instructions may operate on data that has already been
loaded into general purpose data registers **126**, while
other instructions may perform a control operation with
respect to general purpose control registers **128**.

Fetch unit **130** retrieves instructions from

20 instruction cache **122** as necessary, which in turn
retrieves instructions from memory **110** as necessary.
Decode unit **132** decodes instructions to determine basic
information about the instruction, such as instruction
type, source registers, and destination registers.

25 In this example, processor **120** is depicted as an
out-of-order execution processor. Sequencing unit **134**
uses the decoded information to schedule instructions for
execution. In order to track instructions, completion
unit **136** may have data and control structures for storing

30 and retrieving information about scheduled instructions.

As the instructions are executed by execution unit **138**, information concerning the executing and executed instructions is collected by completion unit **136**. Execution unit **138** may use multiple execution subunits.

5 As instructions complete, completion unit **136** commits the results of the execution of the instructions; the destination registers of the instructions are made available for use by subsequent instructions, or the values in the destination registers are indicated as

10 valid through the use of various control flags. Subsequent instructions may be issued to the appropriate execution subunit as soon as its source data is available.

In this example, processor **120** is also depicted as a

15 speculative execution processor. Generally, instructions are fetched and completed sequentially until a branch-type instruction alters the instruction flow, either conditionally or unconditionally. After decode unit **132** recognizes a conditional branch operation,

20 sequencing unit **134** may recognize that the data upon which the condition is based is not yet available; e.g., the instruction that will produce the necessary data has not been executed. In this case, fetch unit **130** may use one or more branch prediction mechanisms in branch

25 prediction unit **140** to predict the outcome of the condition. Control is then speculatively altered until the results of the condition can be determined. Depending on the capabilities of the processor, multiple prediction paths may be followed, and unnecessary

30 branches are flushed from the execution pipeline.

Since speculative instructions can not complete until the branch condition is resolved, many high performance out-of-order processors provide a mechanism to map physical registers to virtual registers. The

5 result of execution is written to the virtual register when the instruction has finished executing. Physical registers are not updated until an instruction actually completes. Any instructions dependent upon the results of a previous instruction may begin execution as soon as

10 the virtual register is written. In this way, a long stream of speculative instructions can be executed before determining the outcome of a conditional branch.

Interrupt control unit **142** controls events that occur during instruction processing that cause execution

15 flow control to be passed to an interrupt handling routine. A certain amount of the processor's state at the time of the interrupt is saved automatically by the processor. After completion of interruption processing, a return-from-interrupt (RFI) can be executed to restore

20 the saved processor state, at which time the processor can proceed with the execution of the interrupted instruction. Interrupt control unit **142** may comprise various data registers and control registers that assist the processing of an interrupt.

25 Certain events occur within the processor as instructions are executed, such as cache accesses or Translation Lookaside Buffer (TLB) misses. Performance monitor **144** monitors those events and accumulates counts of events that occur as the result of processing

30 instructions. Performance monitor **144** is a software-accessible mechanism intended to provide

information concerning instruction execution and data
storage; its counter registers and control registers can
be read or written under software control via special
instructions for that purpose. Performance monitor **144**

5   contains a plurality of performance monitor counters
(PMCs) or counter registers **146** that count events under
the control of one or more control registers **148**. The
control registers are typically partitioned into bit
fields that allow for event/signal selection and

10   accumulation. Selection of an allowable combination of
events causes the counters to operate concurrently; the
performance monitor may be used as a mechanism to monitor
the performance of the stages of the instruction
pipeline.

15   Those of ordinary skill in the art will appreciate
that the hardware in **Figure 1B** may vary depending on the
system implementation. The depicted example is not meant
to imply architectural limitations with respect to the
present invention.

20   With reference now to **Figure 1C**, a prior art diagram
shows software components within a computer system
illustrating a logical relationship between the components
as functional layers of software. The kernel (Ring 0) of
the operating system provides a core set of functions that

25   acts as an interface to the hardware. I/O functions and
drivers can be viewed as resident in Ring 1, while memory
management and memory-related functions are resident in
Ring 2. User applications and other programs (Ring 3)
access the functions in the other layers to perform

30   general data processing. Rings 0-2, as a whole, may be
viewed as the operating system of a particular device.

Assuming that the operating system is extensible, software drivers may be added to the operating system to support various additional functions required by user applications, such as device drivers for support of new
5   devices added to the system.

In addition to being able to be implemented on a variety of hardware platforms, the present invention may be implemented in a variety of software environments. A typical operating system may be used to control program
10   execution within each data processing system. For example, one device may run a Linux® operating system, while another device may run an AIX® operating system.

With reference now to **Figure 1D**, a simple block diagram depicts a typical relationship between software
15   components in a data processing system that is being analyzed in some manner by a trace facility. Trace program **150** is used to analyze application program **151**. Trace program **150** may be configured to handle a subset of interrupts on the data processing system that is being
20   analyzed. When an interrupt or trap occurs, e.g., a single-step trap or a taken-branch trap, functionality within trace program **150** can perform various tracing functions, profiling functions, or debugging functions; hereinafter, the terms tracing, profiling, and debugging
25   are used interchangeably. In addition, trace program **150** may be used to record data upon the execution of a hook, which is a specialized piece of code at a specific location in an application process. Trace hooks are typically inserted for the purpose of debugging,
30   performance analysis, or enhancing functionality.

Typically, trace program **150** generates trace data of various types of information, which is stored in a trace data buffer and subsequently written to a data file for post-processing.

5   Both trace program **150** and application program **151** use kernel **152**, which comprises and/or supports system-level calls, utilities, and device drivers. Depending on the implementation, trace program **150** may have some modules that run at an application-level

10   priority and other modules that run at a trusted, system-level priority with various system-level privileges.

It should be noted that the instruction tracing functionality of the present invention may be placed in a

15   variety of contexts, including a kernel, a kernel driver, an operating system module, or a tracing process or program. Hereinafter, the term "tracing program" or "tracing software" is used to simplify the distinction versus typical kernel functionality and the processes

20   generated by an application program. In other words, the executable code of the tracing program may be placed into various types of processes, including interrupt handlers.

In addition, it should be noted that hereinafter the term "current instruction address" or "next instruction"

25   refers to an instruction within an application that is being profiled/traced and does not refer to the next instruction within the profiling/tracing program. When a reference is made to the value of the instruction pointer, it is assumed that the processor and/or

30   operating system has saved the instruction pointer that was being used during the execution of the application

program; the instruction pointer would be saved into a special register or stack frame, and this saved value is retrievable by the tracing program. Hence, when an instruction pointer is discussed herein, one refers to

5   the value of the instruction pointer for the application program at the point in time at which the application program was interrupted.

With reference now to **Figure 1E**, a diagram depicts typical phases that may be used to characterize the

10   operation of a tracing facility. An initialization phase **155** is used to capture the state of the client machine at the time tracing is initiated. This trace initialization data may include trace records that identify all existing threads, all loaded classes, and all methods for the

15   loaded classes; subsequently generated trace data may indicate thread switches, interrupts, and loading and unloading of classes and jitted methods. A special record may be written to indicate within the trace output when all of the startup information has been written.

20   Next, during the profiling phase **156**, trace records are written to a trace buffer or file. Subject to memory constraints, the generated trace output may be as long and as detailed as an analyst requires for the purpose of profiling or debugging a particular program.

25   In the post-processing phase **157**, the data collected in the buffer is sent to a file for post-processing. During post-processing phase **158**, each trace record is processed in accordance with the type of information within the trace record. After all of the trace records

30   are processed, the information is typically formatted for output in the form of a report. The trace output may be

sent to a server, which analyzes the trace output from processes on a client. Of course, depending on available resources or other considerations, the post-processing also may be performed on the client. Alternatively,

5   trace information may be processed on-the-fly so that trace data structures are maintained during the profiling phase.

As mentioned previously, instruction tracing is an important analysis tool, but instruction tracing is

10   difficult to perform reliably because the act of accessing an instruction to be traced may cause interrupts, thereby causing unwanted effects at the time of the interrupt and generating unwanted trace output information.

15   This type of effect is particularly troublesome to a tracing program that has instruction tracing functionality. At some point in time, the tracing program is given execution control, typically through a single-step or trap-on-branch interrupt. At that point

20   in time, the processor's instruction pointer indicates the next instruction to be executed; the instruction pointer points to the address of the next instruction.

In some cases, the processor may prefetch instructions into an instruction cache. Hence, at the

25   point in time that the single-step or trap-on-branch interrupt occurs, the processor may have a copy of the instruction in a unit within the processor, such as instruction cache **122** or instruction decode unit **132** in **Figure 1B**. Unlike some internal structures in a

30   performance monitor, certain internal structures within the processor are only accessible to the microcode or

nanocode within the processor, and these internal structures are not accessible to application-level code. In other words, there are no processor instructions that can be used by the tracing program to read the

5   processor's copy of the instruction if the processor already has a copy.

In order to perform an instruction tracing operation, the tracing program typically attempts to read the current instruction by using the address that is

10   indicated by the instruction pointer; the instruction pointer points to a location within main memory. However, if the instruction is contained within an execute-only memory block, the attempted access of the instruction by the tracing program causes some type of

15   error signal for which the tracing program should compensate. In other cases, several interrupts can be generated while simply trying to access the instruction if the instruction has not yet been fetched, e.g., interrupts associated with page faults or a TLB miss.

20   Hence, it would be advantageous to provide hardware assistance within a processor in order to capture copies of executed instructions for tracing purposes. The present invention is described in more detail further below with respect to the remaining figures.

25   With reference now to **Figure 2A**, a block diagram depicts an executed-instruction register within a processor that may be used to reveal an executed instruction in accordance with the present invention. Processor **202**, which is similar to processor **120** shown in

30   **Figure 1B**, is constructed to include a special register, executed-instruction register **204**, that contains a copy

of the most recently executed instruction.
Alternatively, executed-instruction register **204** contains
a copy of the opcode of the most recently executed
instruction. Executed-instruction register **204** may be

5   physically placed within various units within processor
202 as appropriate; for example, executed-instruction
register **204** may be contained within an execution unit, a
completion unit, or a performance monitor.
Executed-instruction register **204** may be restricted to

10   read-only operations by application-level code as
necessary.

It should also be noted that the
executed-instruction register need not be a
dedicated-purpose register; the processor may deliver a

15   copy of the most recently executed instruction to a
general purpose register, a performance monitor register,
or other register as may be configured or as may be
appropriate for the implemented processor architecture.
In any case, the executed instruction is made available

20   in a processor structure so that a read operation may be
performed to retrieve a copy of the most recently
executed instruction.

With reference now to **Figure 2B**, a block diagram
depicts an executed-instruction register within a

25   processor that is protected by an executed-instruction
(EI) control flag. Processor **212** contains
executed-instruction register **214** in a manner similar to
Figure 2A. In contrast with **Figure 2A**, **Figure 2B** also
shows control register **216**, which is typically used to

enable features or to specify bit-wise parameters within the processor.

With respect to the present invention, control register **216** also contains executed-instruction (EI) flag **218**, which is a software-specifiable flag that enables the feature of placing the most recently executed instruction into executed-instruction register **214**. The executed-instruction flag may be useful for a variety of reasons. For example, the operation of revealing the most recently executed opcode or instruction may require additional processor time, such as an additional internal cycle, because the processor may have to perform additional work to collect the most recently executed instruction, e.g., obtaining the instruction from one of multiple execution subunits and then delivering it to the executed-instruction register. The executed-instruction flag may be set by a special purpose instruction or by a generalized instruction through which software can read and write control registers and/or status registers within the processor.

With reference now to **Figure 2C**, a flowchart depicts the use of an EI control flag for an executed-instruction register within a processor. The process begins with the completion of the execution of an instruction (step **222**), which may be signaled by a completion unit within the processor. A determination is then made as to whether the executed-instruction (EI) flag has been set (step **224**). If the flag is set, then a copy of the executed instruction or its opcode is written to a register within the processor (step **226**), e.g., a special register that holds only executed instructions, such as

executed-instruction register **214** shown in **Figure 2B**, thereby completing the process.

The flowchart in **Figure 2C** only shows the manner in which a value is written to the executed-instruction

5   register. It is assumed that the executed-instruction register can be read when the executed-instruction or opcode is needed, such as by an instruction tracing routine that writes the executed instruction or opcode to a trace output buffer.

10   With reference now to **Figure 2D**, a block diagram depicts an executed-instruction register that is protected by an EI flag to be used in conjunction with other control flags, such as interrupt control flags. Processor **232** contains executed-instruction register **234**

15   in a manner similar to **Figure 2B**. In contrast with **Figure 2B**, **Figure 2D** shows executed-instruction register **234** as being located within interrupt control unit **236** that contains control register **238** for controlling the operation of the interrupt control unit.

20   Instruction-level tracing is typically accomplished in conjunction with interrupt processing.

A tracing program may set single-step trap flag **240** or taken-branch trap flag **242** that causes the processor to stop executing instructions at appropriate times,

25   e.g., every instruction or every branch-type instruction, respectively. Depending on the specified tasks of the tracing program, the tracing program may or may not need instruction-level information. If so, when the tracing program's single-step interrupt handler or taken-branch

interrupt handler receives control, it needs a copy of the most recently executed instruction.

EI flag **244** may be used to specify that the most recently executed instruction should be copied into an

5   appropriate register, such as executed-instruction register **234**. In this example, EI flag **244** may be ignored when neither single-step trap flag **240** nor taken-branch trap flag **242** have been set. Alternatively, a copy of the executed instruction may be placed into the

10   executed instruction register whenever single-step trap flag **240**, taken-branch trap flag **242**, or other interrupt-enable flag has been set without regard to an executed-instruction flag.

It should be noted that the term "most recently

15   executed instruction" refers to an instruction associated with application-level processing; the execution of instructions within an interrupt handler would usually not be of interest to a profiling program. Hence, the described examples may have a feature in which the use of

20   an executed-instruction register may be suspended while interrupts are being serviced.

With reference now to **Figure 3A**, a block diagram depicts a taken-branch instruction buffer to be used to store executed instructions with respect to the most

25   recent branch-type instruction. The example shown in **Figure 3A** shows processor **302** that contains an interrupt control unit that is similar to the processor shown in **Figure 2D**. As described above with respect to **Figure 2D**, a tracing program may set a taken-branch trap flag that

30   causes the processor to stop executing instructions at

it returns execution control. Hence, a
return-from-interrupt (RFI) from either the responsible
handler or a taken-branch trap handler allows the
processor to reset the start indicator and the end
5 indicator.

   With reference now to **Figure 3B**, a flowchart depicts
the use of a taken-branch instruction buffer within a
processor. The process shown in **Figure 3B** is similar to
the process shown in **Figure 2C** except that the operation
10 of a taken-branch instruction buffer has been
incorporated into the flowchart shown in **Figure 3B**. The
process begins with the completion of the execution of an
instruction (step **312**), which may be signaled by a
completion unit within the processor. A determination is
15 then made as to whether the executed-instruction (EI)
flag has been set (step **314**). If not, then process is
complete. If the flag is set, then a copy of the
executed instruction or its opcode is written to a
register within the processor (step **316**). In this
20 example, the executed-instruction register and the
taken-branch instruction buffer operate simultaneously.

   A determination is then made as to whether the
taken-branch flag is set (step **318**). If not, then the
process is complete. If the taken-branch flag is set,
25 then a copy of the executed instruction or its opcode is
written to a taken-branch instruction buffer within the
processor (step **320**). A determination is then made as to
whether the taken-branch instruction buffer is full (step
322). If not, then the process is complete. If the
30 taken-branch instruction buffer is full, then the buffer

full flag is set (step **324**), and the processor generates
a buffer-full interrupt (step **326**) in an attempt to empty
the buffer, after which the process is complete.

Figure 3B shows only the operation of filling and
5   monitoring the taken-branch instruction buffer. It is
assumed that when the taken-branch trap handler is
invoked after the execution of a branch instruction (not
shown in **Figure 3B**), the taken-branch trap handler would
write the saved instructions or opcodes to a trace output
10   buffer.

With reference now to **Figure 4**, a block diagram
depicts an alternative embodiment for a taken-branch
instruction buffer to be used to store executed
instructions. The example shown in **Figure 4** shows
15   processor **402** that contains an interrupt control unit
that is similar to the processor shown in **Figure 3A**.
However, it may be assumed that the size of the
taken-branch instruction buffer shown in **Figure 3A** is
limited by the fact that the buffer is located within the
20   processor. A relatively small taken-branch instruction
buffer would be problematic for a series of instructions
between branch-type instructions in which the number of
instructions between those branch-type instructions was
greater than the buffer size.

25   In contrast to **Figure 3A**, processor **402** does not
contain the taken-branch instruction buffer within the
processor. Instead, taken-branch instruction buffer
pointer **404** points to a location in memory where the
taken-branch instruction buffer can be found. When
30   appropriate, the processor writes a copy of an

instruction or its opcode to the taken-branch instruction buffer. Configuration registers **406** and **408** may hold the size of the taken-branch instruction buffer and the next unused entry offset indicator, respectively. By locating

5 the taken-branch instruction buffer in main memory, there should be adequate storage space for saving any series of instructions between branch-type instructions in application-level software.

When a taken-branch trap handler is invoked after

10 the execution of a branch instruction, the taken-branch trap handler would write the saved instructions or opcodes from the taken-branch instruction buffer to a trace output buffer. Alternatively, the taken-branch instruction buffer may be considered to be one of a

15 plurality of trace output buffers, and the taken-branch trap handler may merely save the taken-branch instruction buffer to persistent storage when appropriate rather than transferring its contents to another trace output buffer.

In this example, it may be assumed that the

20 taken-branch instruction buffer is always filled from the beginning of the buffer after it is emptied or reset. In a manner similar to **Figure 3B**, full flag **410** is associated with the buffer, and the full flag is set when the buffer is full, thereby causing an appropriate

25 handler to be invoked to remedy the full condition. A return-from-interrupt may be used by the processor to reset the next unused entry offset indicator.

For the embodiment shown in **Figure 4**, the write operation to store the copy of the instruction or opcode

30 in a taken-branch instruction buffer in memory may cause unwanted interruptions, such as page faults. Hence, some

preparation may be necessary to ensure that such interruptions do not occur. One methodology for preventing these interruptions is discussed in the following copending and commonly assigned application

5 entitled "METHOD AND SYSTEM FOR INSTRUCTION TRACING WITH ENHANCED INTERRUPT AVOIDANCE", U.S. Application Serial Number _____, Attorney Docket Number AUS920010716US1, filed on _____, currently pending, herein incorporated by reference.

10 As described therein, during an initialization phase, the tracing program allocates a taken-branch instruction buffer in physical memory, maps the buffer to its virtual address space, and pins the buffer. At any subsequent point in time, data accesses to the

15 taken-branch instruction buffer would use non-virtual addressing, i.e., physical addressing instead of virtual addressing. More specifically, the physical address would be registered into the taken-branch instruction buffer pointer. By using non-virtual addressing, there

20 is no opportunity for TLB misses, which could occur when addressing the buffer via virtual addressing. In a particular embodiment that uses the Intel® IA-64 processor architecture, the "dt" bit (data address translation bit) of the processor status register

25 ("psr.dt") can be used to control virtual addressing versus physical addressing. When the "psr.dt" bit is set to "1", virtual data addresses are translated; when the "psr.dt" bit is set to "0", data accesses use physical addressing.

30 In an alternative embodiment using the Intel® IA-64 processor architecture, TLB misses can be avoided by

using a translation register. Translation registers are managed by software, and once an address translation is inserted into a translation register, it remains in the translation register until overwritten or purged.

5   Translation registers are used to lock critical address translations; all memory references made to a translation register will always hit the TLB and will never cause a page fault. With respect to the present invention, a translation register could be configured for the

10   taken-branch instruction buffer during the initialization phase of the tracing software, thereby ensuring that there are no TLB misses for the taken-branch instruction buffer.

Figures 2A-4 depict various mechanisms within a

15   processor for revealing the most recently executed instruction; after the instruction is completed, the opcode of the instruction or the entire instruction is revealed in one of a variety of manners, such as by writing the opcode or instruction to a register that may

20   be read by application-level code. These mechanisms eliminate various types of interrupts that may occur when attempting to read a copy of an instruction, thereby avoiding the creation of any additional problems with those interrupts.

25   However, some advanced processor architectures incorporate other features that create additional difficulties with instruction tracing, such as predication, which is the conditional execution of an instruction based on a qualifying predicate. If a

30   processor implements predication functionality, then typically most processor instructions can be guarded by a

qualifying predicate, which is typically a predicate register whose value determines whether the processor commits the results computed by the qualified instruction.

5   Predicate registers are usually one-bit values in which a value of "zero" is interpreted as false and in which a value of "one" is interpreted as true. If the predicate is true, then the instruction executes completely or fully; if the predicate is false, then the

10   instruction does not execute fully because it does not modify the state of the processor in a way that would affect the execution of subsequent instructions, although there are some instructions in the Intel® IA-64 processor architecture that do not operate in this manner. In

15   other words, if the predicate is false, then the predicated instruction's architectural updates are suppressed, and the instruction behaves like a so-called "nop", which is an abbreviated term for a "no-op" or a "no operation".

20   Predication is particularly useful because predicated instructions can be used for conditional execution of branch-type instructions, which results in longer series of unbranched instruction streams and the elimination of hardware-resource-consuming misprediction

25   penalties. In essence, predication allows a compiler to convert control dependencies into data dependencies, thereby allowing the compiler to optimize instruction scheduling during compilation. For example, predicate registers are generally used in pairs, with one predicate

30   register having the complement of the value of the other predicate register of the pair. Predication allows a

processor to execute two execution paths in parallel in which a first predicate register "switches off" a first instruction branch while a second predicate register "switches on" a second instruction branch. However,

5   n-way branching can also be controlled with predicate registers.

However, predication creates additional difficulties for instruction tracing because the instruction tracing software should be able to identify which of the traced

10 instructions in the trace output data were fully executed. Hence, it would be advantageous to provide hardware assistance within a processor in order to capture information to assist in the identification of partially executed instructions versus fully executed

15 instructions in conjunction with other functionality within the instruction tracing software. In other words, a hardware mechanism would assist in identifying the manner in which instructions were disposed, i.e., as non-committed instructions versus committed instructions.

20   The features of the present invention concerning instruction disposition are described in more detail further below with respect to **Figures 5B-6C**.

With reference now to **Figure 5A**, a prior art diagram depicts the predicate register file in the Intel® IA-64

25 architecture. Predicate registers are implemented in a variety of processor architectures, such as the general class of computers that uses the term Explicitly Parallel Instruction Computer (EPIC), of which the Intel® IA-64 architecture is merely one example. While some of the

30 examples herein may explicitly refer to the IA-64 architecture, the tracing features of the present

invention may be implemented on a variety of processor architectures.

In order to support the ability to execute numerous instructions in parallel, the IA-64 architecture is massively resourced with a large number of general and special purpose registers that enable multiple computations to be performed without having to frequently save and restore intermediate data to and from memory. An IA-64 compiler has several primary register stacks available: 128 64-bit general-purpose registers that are used to hold values for integer and multimedia computations; 128 82-bit floating-point registers that are used for floating-point computations; 8 64-bit branch registers that are used to specify the target addresses of indirect branches; and 64 one-bit predicate registers (PR0-PR63) that control conditional execution of instructions and conditional branches. **Figure 2A** shows predicate registers (PR0-PR63) that may be used by an Intel® IA-64 compiler to predicate the execution of instructions. In the prior art, all 64 registers are available for general use, although the first register PR0 is read-only and always reads as a "one" or "true".

With reference now to **Figure 5B**, a block diagram depicts an executed-instruction register and a disposition trace register. Processor **502**, which is similar to processor **202** shown in **Figure 2A**, is constructed to include a first dedicated register, executed-instruction register **504**, that contains a copy of the most recently executed instruction or its opcode,

and a second dedicated register, disposition trace register **506**, which may be a one-bit indicator flag.

The disposition trace register indicates the manner in which the most recently executed instruction was
5   disposed. If the results of the most recently executed instruction were not committed, i.e., the instruction was only partially executed, then the disposition flag may be cleared or may remain clear. If the results of the most recently executed instruction were committed, i.e., the
10   instruction was fully executed, then the disposition flag may be set. Alternatively, the meaning that is attached to the values of the flag may be reversed.

As discussed above, instructions can be associated with a predicate register, and the value of the predicate
15   register determines whether the results of the associated instruction are committed. Assuming that the meaning of the values of a predicate register are equivalent to the meaning of the values of the disposition trace register, then the value of the predicate register that is
20   associated with the most recently executed instruction may be written directly to the disposition trace register. For example, if a non-zero value in a predicate register indicates that the results of the instruction should be committed, and if a non-zero value
25   in the disposition trace register indicates that the most recently executed instruction was fully executed, then the value of the predicate register can be transferred directly to the disposition trace register. In this manner, the disposition trace register would indicate
30   whether the most recently executed instruction was

partially executed or fully executed as controlled by the predicate register.

It should be noted that, in some processor architectures, a subset of instructions cannot be predicated or the predication bits in the instruction opcode are ignored. For this subset of instructions, the instructions would always be considered to have been fully executed. Hence, the determination of the appropriate value for the disposition trace register is considerably easier because the disposition trace register would be directly set to a non-zero value without reference to a predicate register when one of these instructions has been executed. For example, a disposition flag could be set during the completion stage of one of these non-predicated instructions with an appropriate signal from the instruction's completion unit or execution unit to the circuitry of the disposition flag. The following examples assume that all instructions within the processor may be predicated, although it should be noted that the examples could be modified to include special-case processing for the simpler cases in which the results of an instruction are always committed.

Registers **504** and **506** may be physically placed within various units within processor **502** as appropriate, such as an execution unit, a completion unit, or a performance monitor. Alternatively, disposition trace register **506** may exist independently of an executed-instruction register or without pairing it with an executed-instruction register. Registers **504** and **506** may be restricted to read-only operations by

application-level code as necessary. It should also be noted that registers **504** and **506** need not be dedicated registers, e.g., they may be general purpose registers, performance monitor registers, or other register types as may be configured or as may be appropriate for the implemented processor architecture. In any case, the executed instruction and its disposition are made available in a processor structure so that a read operation may be performed to retrieve a copy of the most recently executed instruction and information about the manner in which it was disposed.

With reference now to **Figure 5C**, a block diagram depicts a disposition trace register within a processor that is protected by a disposition trace (DT) control flag. Processor **512** contains disposition trace register **514** in a manner similar to **Figure 5B**. In contrast with **Figure 5B**, **Figure 5C** also shows control register **516**, which is typically used to enable features or to specify bit-wise parameters within the processor.

With respect to the present invention, control register **516** also contains disposition trace (DT) flag **518**, which is a software-specifiable flag that enables the feature of recording the disposition of the most recently executed instruction into disposition trace register **514**. The disposition trace flag may be useful for a variety of reasons. For example, the operation of revealing the disposition of the most recently executed instruction may require additional processor time, such as an additional internal cycle, because the processor may have to perform additional work to determine the disposition of the most recently executed instruction,

e.g., obtaining a predicate value from one of the predicate registers and then delivering it to the disposition trace register. The disposition trace flag may be controlled by a special purpose instruction or by

5   a generalized instruction through which software can read and write control registers and/or status registers within the processor.

With reference now to **Figure 5D**, a flowchart depicts the use of a DT control flag for a disposition trace

10   register within a processor. The process begins with the completion of the execution of an instruction (step **522**), which may be signaled by a completion unit within the processor. A determination is then made as to whether the disposition trace (DT) flag has been set (step **524**).

15   If the flag is set, then a copy of the value in the predicate register that is associated with the executed instruction is written to a register within the processor (step **526**), e.g., a dedicated register that holds only the most recently used predicate value, such as

20   disposition trace register **514** shown in **Figure 5C**, thereby completing the process. (The predicate register that is associated with an instruction may be indicated by a set of bits in the instruction's opcode that are dedicated to that purpose.) Alternatively, the

25   completion unit's signal is directed to the disposition register to reflect that the instruction was fully executed.

The flowchart in **Figure 5D** only shows the manner in which a value is written to the disposition trace

30   register. It is assumed that the disposition trace register can be read when the disposition indicator is

needed, such as by a routine within tracing software that writes the disposition indicator to a trace output buffer in conjunction with other information.

With reference now to **Figure 6A**, a block diagram

5   depicts a multi-bit disposition trace register to be used to store disposition indicator values with respect to a series of recently executed instructions. The example shown in **Figure 6A** shows processor **602** that is similar to the processor shown in **Figure 5C**. It may be assumed that

10   the disposition trace register that is shown in **Figure 5C** is a one-bit register. In contrast with **Figure 5C**, **Figure 6A** shows multi-bit disposition register **604** for storing a plurality of disposition indicator values. This example also illustrates that a disposition trace

15   register can be used independently of an executed-instruction register.

Control register **606** also contains DT flag **608**, which is a software-specifiable flag that enables or disables the disposition tracing feature. As most

20   instructions are executed, the processor places a copy of the predicate value that is associated with the most recently executed instruction into the disposition trace register and advances the pointers associated with the disposition trace register. In this example, the

25   multi-bit disposition trace register is filled in a rotating or wrap-around manner, and start indicator **610** and end indicator **612** are used to point to the first and last entries in the disposition trace register. If the disposition trace register is a 128-bit register,

30   indicators **610** and **612** would be 7 bit values. As in

Figure 5C, a DT flag may be used to qualify, i.e., enable or disable, the operation of the disposition register. If either an execution-instruction register, such as register **514** in **Figure 5C**, or a taken-branch instruction

5   buffer, such as **buffer 304** in **Figure 3A**, are simultaneously used with a disposition trace register, then the disposition trace register may operate in parallel with either of those other structures.

To prevent register overflow, full flag **614** is

10   associated with the disposition trace register. When the disposition trace register is full, the full flag is set, and the appropriate interruption handler is called by the processor in an attempt to empty the register; the processor may assume that the appropriate handler has

15   emptied the register when it returns execution control. Hence, a return-from-interrupt (RFI) allows the processor to reset the start indicator and the end indicator. Alternatively, the register may be filled starting with the least-significant bit, thereby requiring only one

20   fill indicator with the disposition trace register to indicate the position of the next unused bit within the disposition trace register.

With reference now to **Figure 6B**, a flowchart depicts the use of a multi-bit disposition trace register within

25   a processor. The process shown in **Figure 6B** is similar to the process shown in **Figure 5D** except that the operation of a multi-bit disposition trace register has been incorporated into the flowchart shown in **Figure 6B**. The process begins with the completion of the execution

30   of an instruction (step **622**), which may be signaled by a

completion unit within the processor. A determination is
then made as to whether the disposition trace (DT) flag
has been set (step **624**). If not, then process is
complete. If the flag is set, then a copy of the
5   predicate value for the executed instruction is written
to a register within the processor (step **626**), and the
position indicator or indicators for the disposition
trace register is/are updated (step **628**).

A determination is then made as to whether the
10   disposition trace register is full (step **630**). If not,
then the process is complete. If the disposition trace
register is full, then the full flag is set (step **632**),
and the processor generates a register-full interrupt
(step **634**) in an attempt to empty the register, after
15   which the process is complete.

Figure 6B shows only the operation of filling and
monitoring the disposition trace register. It is assumed
that when the interruption handler is invoked for the
register-full interrupt, the interruption handler would
20   write the saved disposition values to a trace output
buffer.

With reference now to **Figure 6C**, a block diagram
depicts an alternative embodiment for a disposition trace
buffer to be used to store disposition values. The
25   example shown in **Figure 6C** shows processor **652** that is
similar to the processor shown in **Figure 6A**; control
register **654** contains PT flag **656**. However, it may be
assumed that the size of the disposition trace register
shown in **Figure 6A** is limited by the fact that the
30   register is located within the processor. A relatively

AUS920010717US1

42

small disposition trace register would be problematic for a long series of instructions in which the number of instructions and their associated disposition values was greater than the register size.

5   In contrast to **Figure 6A**, processor **652** does not contain a disposition trace register within the processor. Instead, a disposition trace buffer pointer **658** points to a location in memory where a disposition trace buffer can be found. When appropriate, the

10   processor writes a copy of the predicate value that is associated with the most recently executed instruction to the disposition trace buffer. Configuration registers **660** and **662** may hold the size of the disposition trace buffer and the next unused entry offset indicator,

15   respectively. By locating the disposition trace buffer in main memory, there should be adequate storage space for saving the disposition values associated with any series of instructions in application-level software.

      In this example, it may be assumed that the

20   disposition trace buffer is always filled from the beginning of the buffer after it is emptied or reset. In a manner similar to **Figure 6A**, full flag **664** is associated with the buffer, and the full flag is set when the buffer is full, thereby causing an appropriate

25   handler to be invoked to remedy the full condition. A return-from-interrupt may be used by the processor to reset the next unused entry offset indicator.

      For the embodiment shown in **Figure 6C**, the write operation to store the copy of the most recently used

30   predicate value in a disposition trace buffer in memory may cause unwanted interruptions, such as page faults.

Hence, some preparation may be necessary to ensure that such interruptions do not occur. Methodologies for preventing these interruptions are discussed above with respect to **Figure 4**.

5   The advantages of the present invention should be apparent in view of the detailed description of the invention that is provided above. A special mechanism is provided within the processor for revealing the disposition of the most recently executed instruction.

10   For most instructions, after the instruction is completed, the predicate value that is associated with the most recently executed instruction is revealed in one of a variety of manners, such as by writing the predicate value to a register that may be read by application-level

15   code. For a subset of instructions, the register merely indicates that the instruction was fully executed without regard to any predicate values. Instruction tracing software can subsequently use the trace information to determine which instructions were fully executed.

20   It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that some of the processes associated with the present invention are

25   capable of being distributed in the form of instructions in a computer readable medium and a variety of other forms, regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media

30   include media such as microcode, nanocode, EPROM, ROM, tape, paper, floppy disc, hard disk drive, RAM, and

CD-ROMs and transmission-type media, such as digital and analog communications links.

The description of the present invention has been presented for purposes of illustration but is not

5   intended to be exhaustive or limited to the disclosed embodiments. Many modifications and variations will be apparent to those of ordinary skill in the art. The embodiments were chosen to explain the principles of the invention and its practical applications and to enable

10   others of ordinary skill in the art to understand the invention in order to implement various embodiments with various modifications as might be suited to other contemplated uses.